



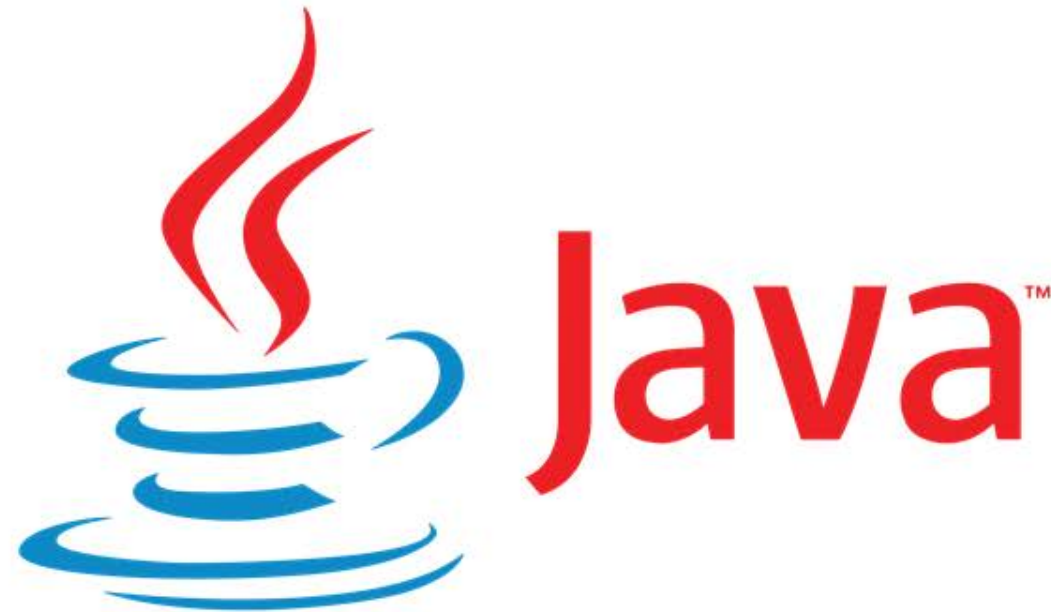
جامعة الأمير سطام بن عبد العزيز
Prince Sattam Bin Abdulaziz University

كلية هندسة وعلوم الحاسب
College of Computer Engineering and Sciences



جامعة الامير سطام بن عبدالعزيز
Prince Sattam Bin Abdulaziz University

نادي الحاسب
Computer Club



Object Oriented Programming

Object Oriented Programming

Lecture 01

What IS OOP ?

What IS Object Oriented Programming ?

- **Object-oriented programming (OOP)** is a [programming paradigm](#) based on the concept of "[objects](#)"
- A [programming paradigm](#) : is a style of programming, a way of thinking about software construction.
- A programming paradigm does not refer to a specific language but rather to a way to build a program or a methodology to apply.
- Some languages make it easy to write in some paradigms but not others.
- Some Programming Languages allow the programmer to apply more than one Paradigm.

Programming Paradigms

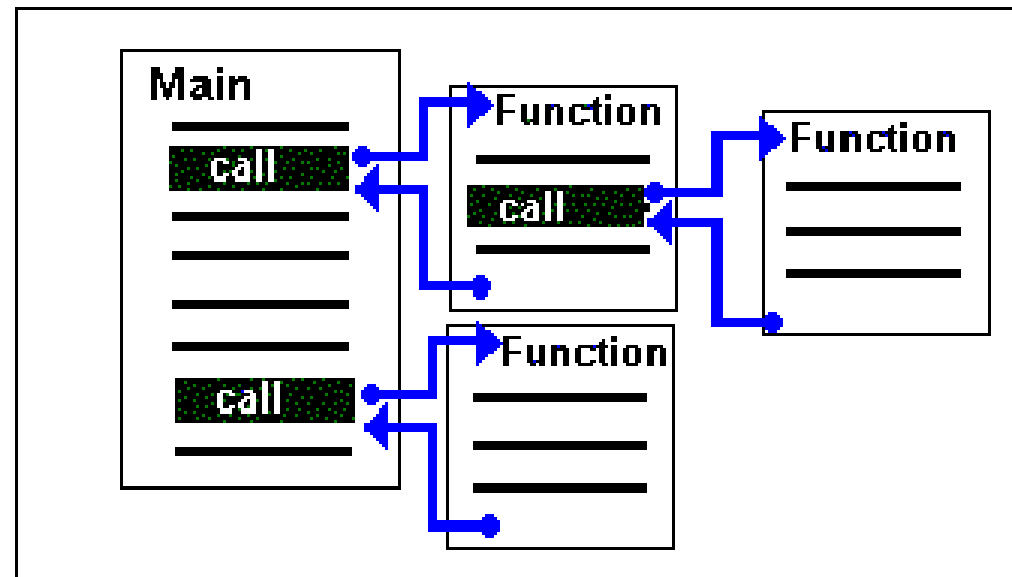
- The programming paradigm refers to a way of conceptualizing and structuring the tasks a computer performs.

Paradigm	Languages	Description
Procedural	BASIC, Pascal, COBOL, FORTRAN, Ada	Emphasizes linear steps that provide the computer with instructions on how to solve a problem or carry out a task
Object-oriented	Smalltalk, C++, Java	Formulates programs as a series of objects and methods that interact to perform a specific task
Declarative	Prolog	Focuses on the use of facts and rules to describe a problem
Functional	LISP, Scheme, Haskell	Emphasizes the evaluation of expressions, called functions
Event-driven	Visual Basic, C#	Focuses on selecting user interface elements and defining event-handling routines that are triggered by various mouse or keyboard activities

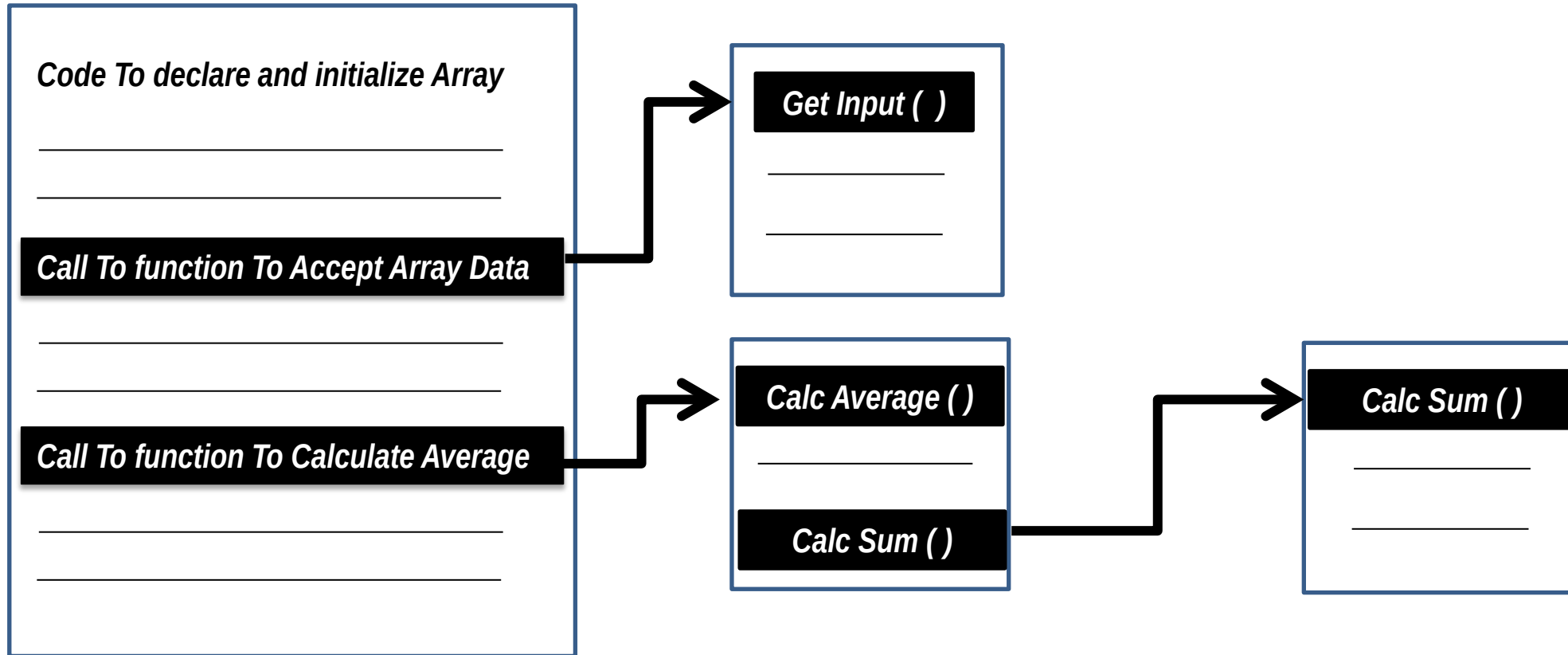
Example of Previous Programming Paradigm

Procedural Programming

Procedural programming (PP), also known as inline programming takes a top-down approach. It is about writing a list of instructions to tell the computer what to do step by step. It relies on procedures or routines.



Procedural Programming Example : Program to Calculate Average of Array Items



- Object-oriented programming (OOP) is a [programming paradigm](#) based on the concept of "[objects](#)"

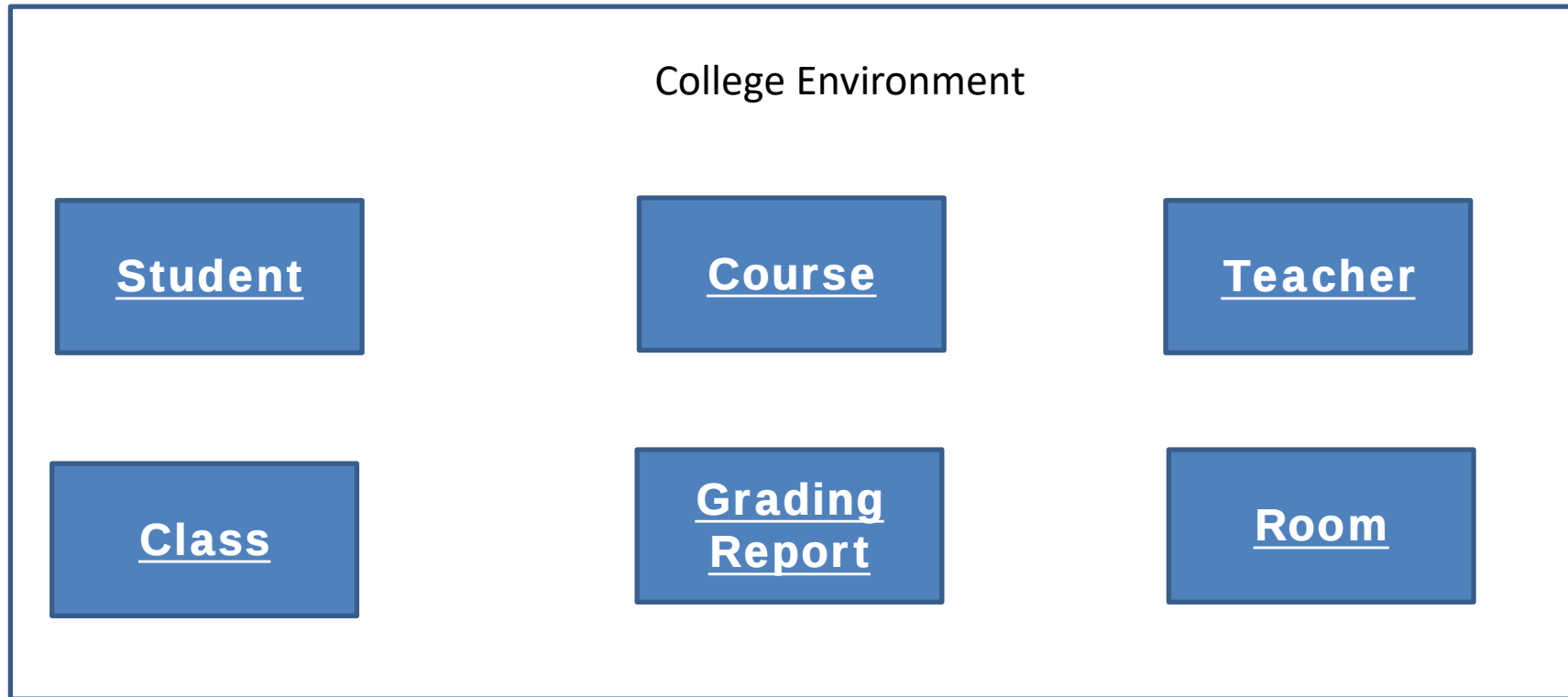
Object : is a thing (Tangible – Intangible)



Objects in Student housing management Program



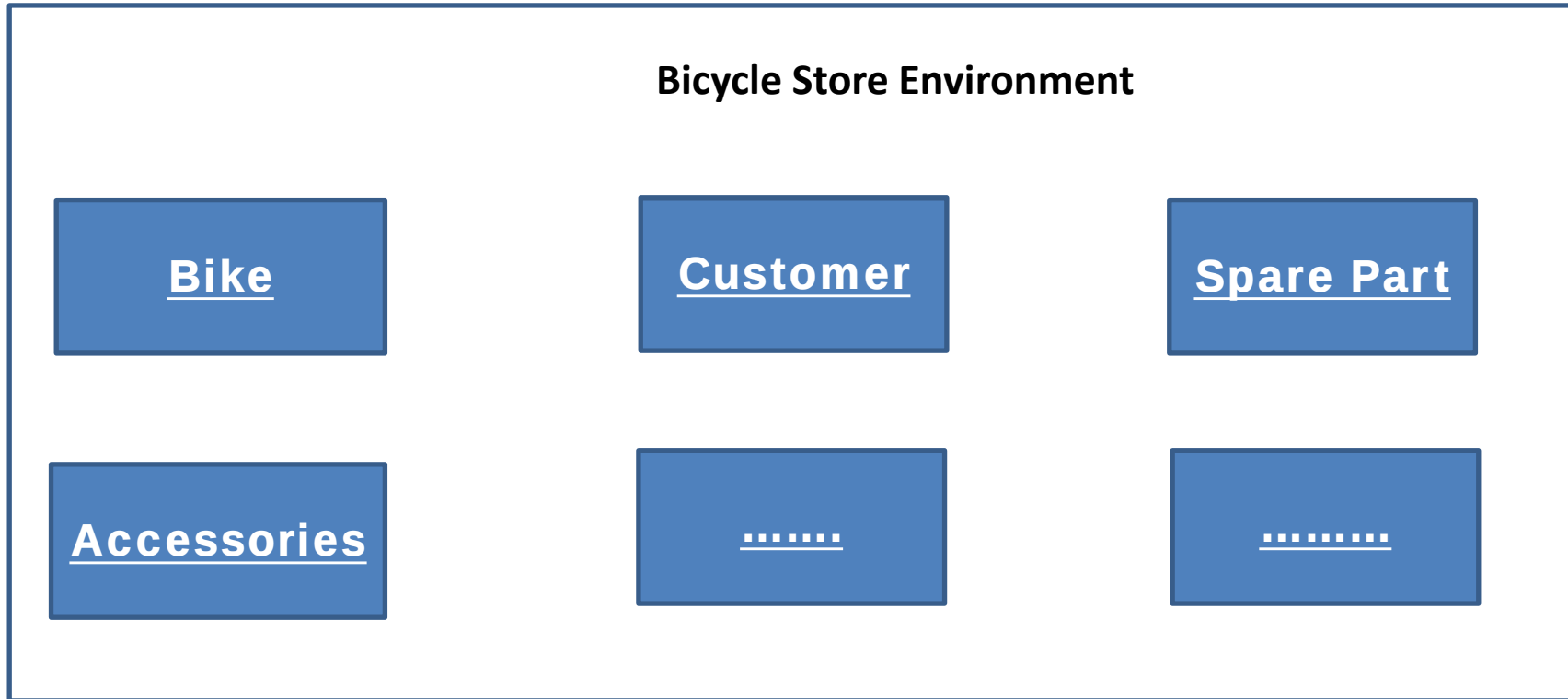
Objects in College Management Program



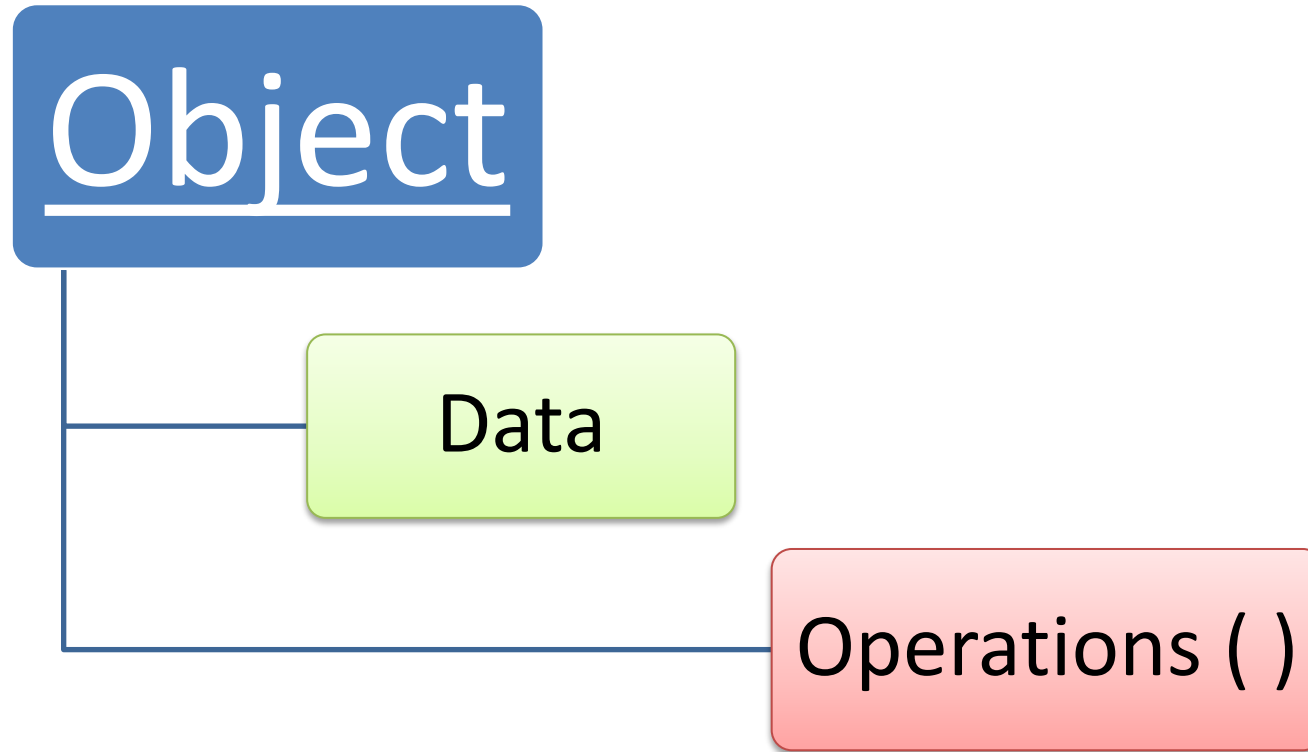
Objects in Super market Program



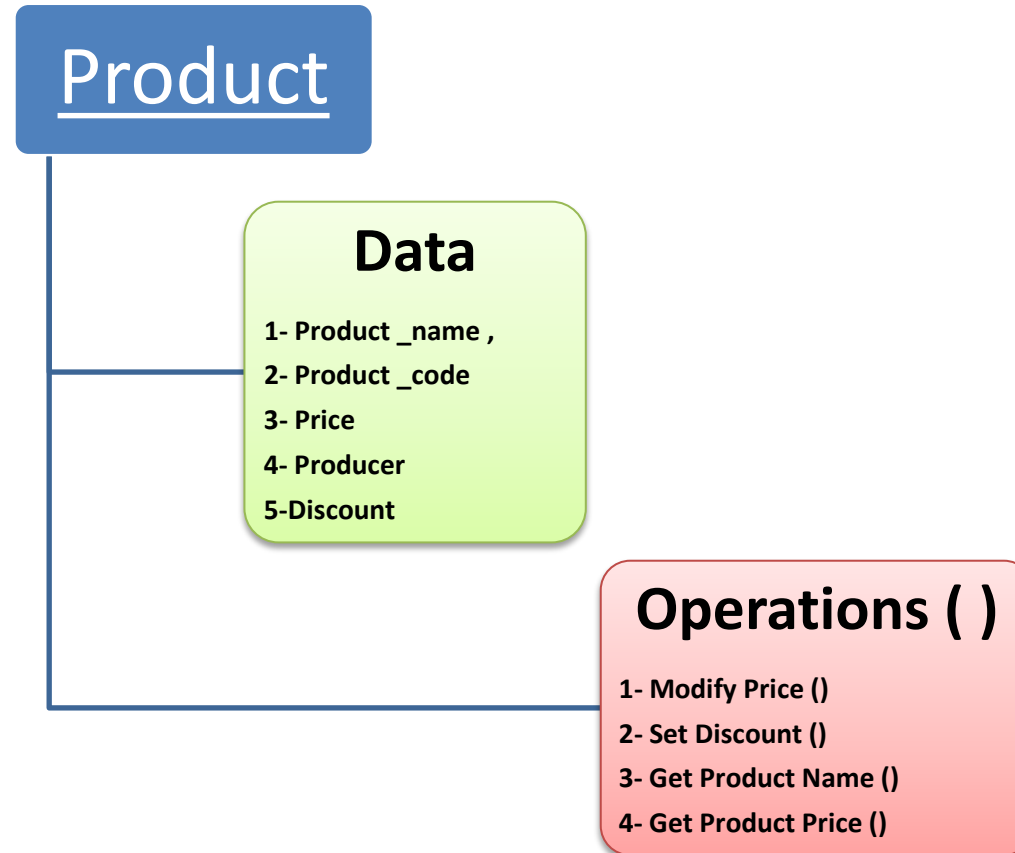
Objects in Bicycle Store Program



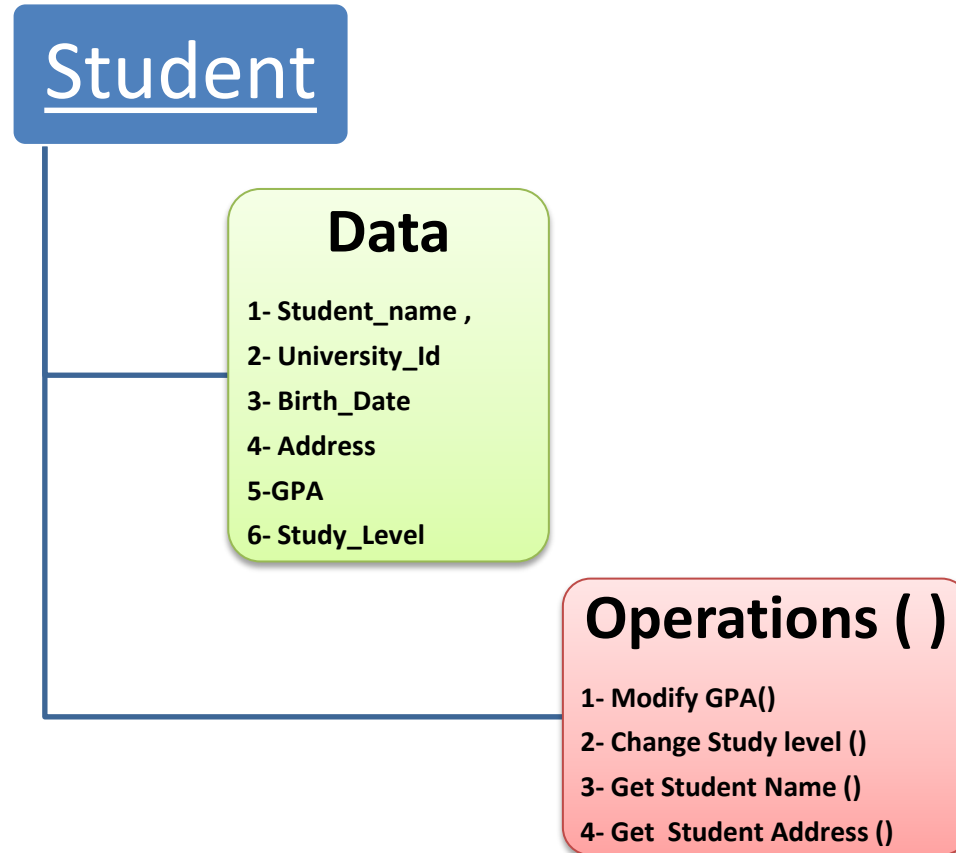
Object Is comprised Of ?



Object Is comprised Of ?



Object Is comprised Of ?

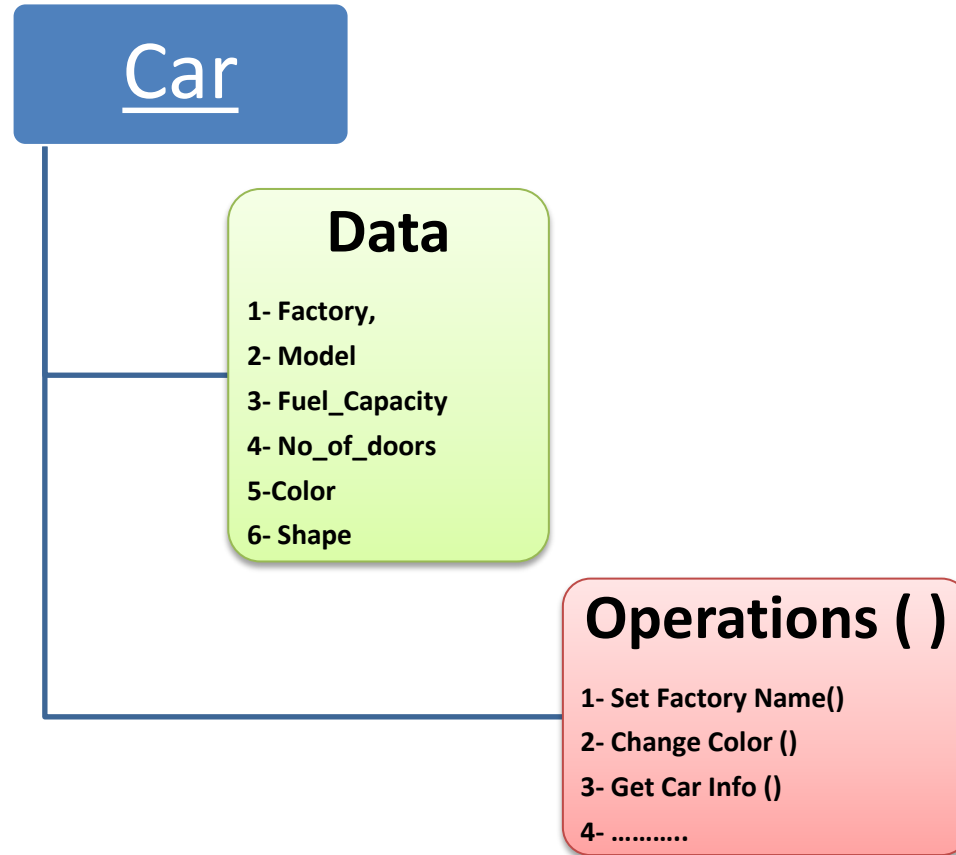


Object Oriented Programming

Lecture 01 – Part 3

What IS OOP ?

Object Is comprised Of ?



What is Class ? Why we need It ?

Student 1

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is Class ? Why we need It ?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is Class ? Why we need It ?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

What is Class ? Why we need It ?

Student 1

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

= Ahmed

= 1050

=3.75

= 5

Objects and Classes

- Classes: Where Objects Come From
 - A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's fields), and the actions that an object can perform (the object's methods).
 - You can think of a class as a code "blueprint" that can be used to create a particular type of object.

Objects and Classes

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an *instance* of the class.

Objects and Classes

Example:

This expression creates a
Scanner object in memory.

```
Scanner keyboard = new Scanner(System.in);
```

The object's memory address
is assigned to the keyboard
variable.



Objects and Classes

Example:

This expression creates a
Random object in memory.

Random rand = new Random();

The object's memory address is
assigned to the rand variable.



Writing a Class, Step by Step

- A Rectangle object will have the following fields:

Rectangle
length width
setLength() setWidth() getLength() getWidth() getArea()

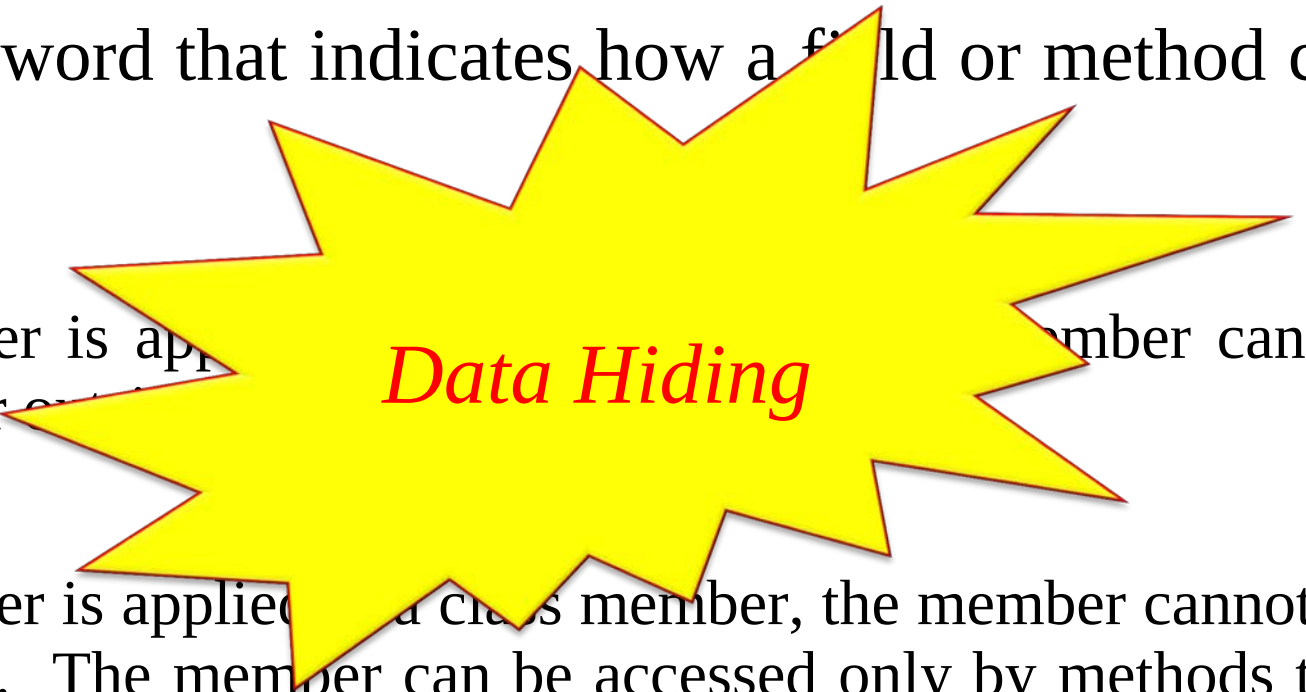
Writing the Code

```
public class Rectangle  
{  
    private double length;  
    private double width;  
}
```

Rectangle
length width
setLength() setWidth() getLength() getWidth() getArea()

Access Modifiers

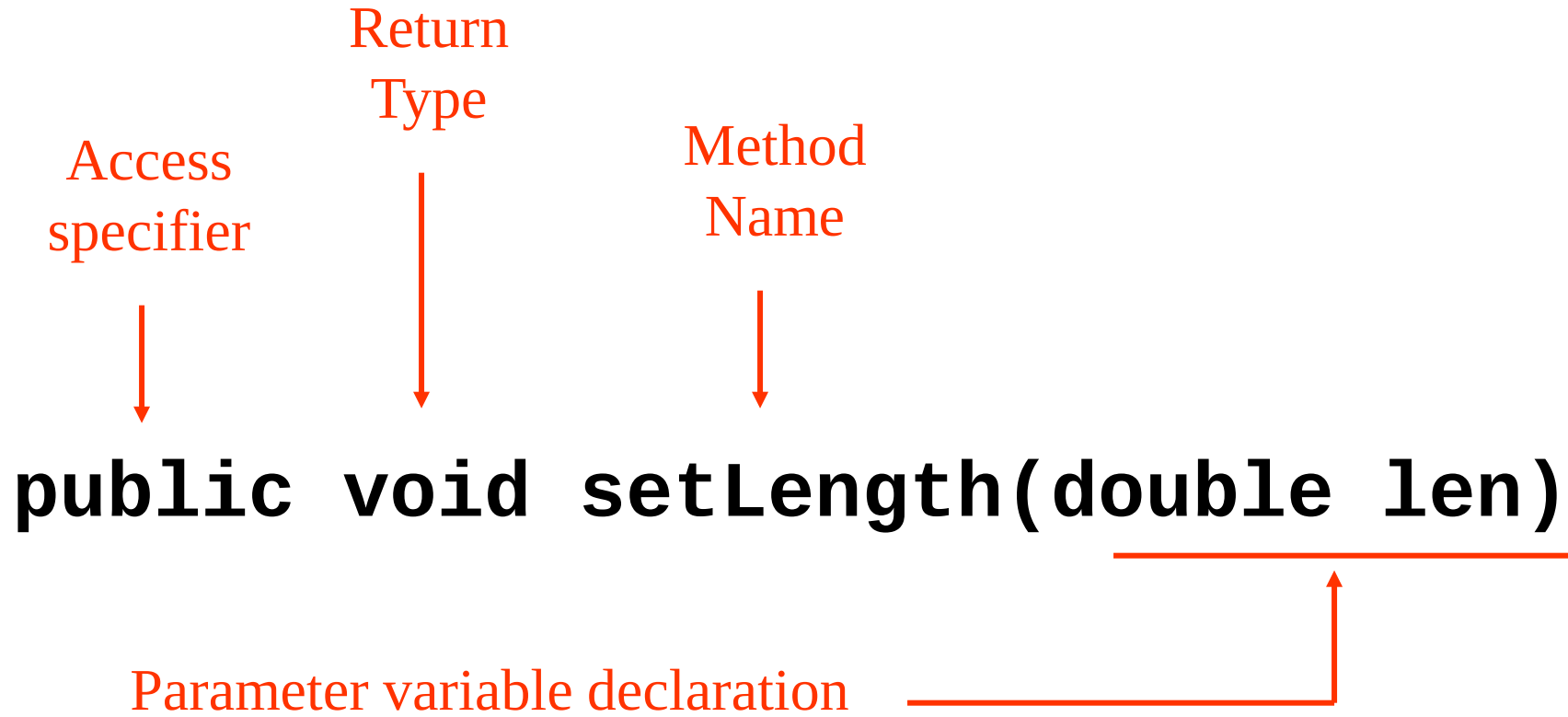
- An access modifier is a Java keyword that indicates how a field or method can be accessed.
- **public**
 - When the `public` access modifier is applied to a class member, the member can be accessed by code inside the class or outside the class.
- **private**
 - When the `private` access modifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.



Data Hiding

Data Hiding

- An object hides its internal, private fields from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and change the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.
- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.



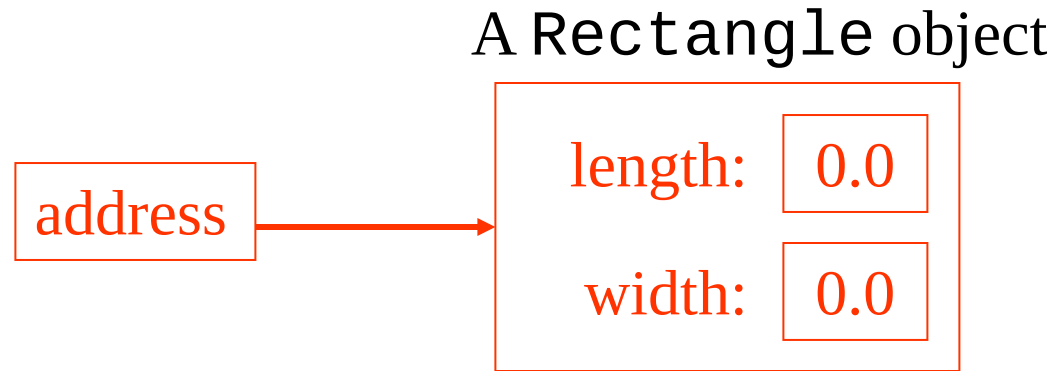
Rectangle
- width : double - length : double
+ setWidth(w : double) : void + setLength(len : double): void + getWidth() : double + getLength() : double + getArea() : double

```
public class Rectangle
{
    private double length;
    private double width;
    public void setLength(double len)
    {
        length = len;
    }
}
```

Creating a Rectangle object

```
Rectangle box = new Rectangle ();
```

The box
variable holds
the address of
the Rectangle
object.

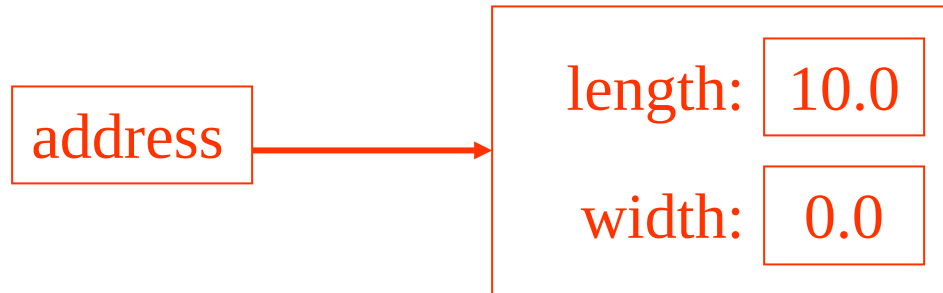


Calling the setLength Method

```
box.setLength(10.0);
```

The box
variable holds
the address of
the
Rectangle
object.

A Rectangle object



This is the state of the box object after
the `setLength` method executes.

Writing the getLength Method

```
public class Rectangle
{
    private double length;
    private double width;
    public void setLength(double len)
    {
        length = len;
    }
    public double getLength()
    {
        return length;
    }
}
```

```
public class Rectangle
{
    private double width;
    private double length;

    public void setWidth(double w)
    {
        width = w;
    }
    public void setLength(double len)
    {
        length = len;
    }
    public double getWidth()
    {
        return width;
    }
    public double getLength()
    {
        return length;
    }
    public double getArea()
    {
        return length * width;
    }
}
```

Instance Fields and Methods

- Fields and methods that are declared as previously shown are called *instance fields* and *instance methods*.
- Objects created from a class each have their own copy of instance fields.
- Instance methods are methods that are not declared with a special keyword, `static`.

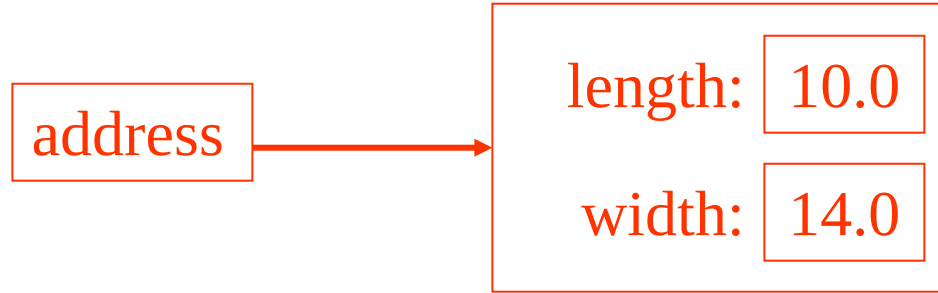
Instance Fields and Methods

- Instance fields and instance methods require an object to be created in order to be used.
- For example, every room can have different dimensions.

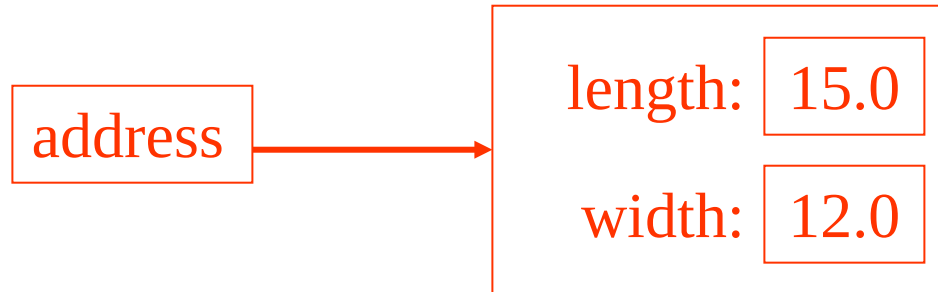
```
Rectangle kitchen = new Rectangle();  
Rectangle bedroom = new Rectangle();  
Rectangle den = new Rectangle();
```

States of Three Different Rectangle Objects

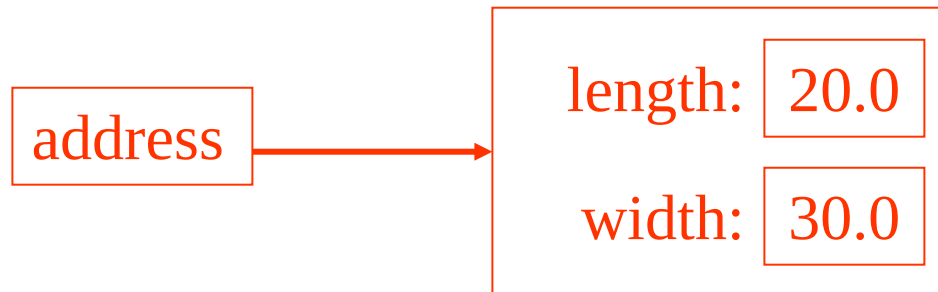
The `kitchen` variable holds the address of a `Rectangle` Object.



The `bedroom` variable holds the address of a `Rectangle` Object.



The `den` variable holds the address of a `Rectangle` Object.



Object Oriented Programming

Lecture 02 – Part 4

Create Your First Class

Accessors and Mutators

```
public class Rectangle
```

```
{
```

```
    private double width;
```

```
    private double length;
```

```
    public void setWidth(double w)
```

```
    {    width = w;
```

```
    }
```

```
    public void setLength(double len)
```

```
    {    length = len;
```

```
    }
```

```
    public double getWidth()
```

```
    {    return width;
```

```
    }
```

```
    public double getLength()
```

```
    {    return length;
```

```
    }
```

```
    public double getArea()
```

```
    {    return length * width;
```

```
    }
```

```
}
```



Setter , Mutator



Getter, Accessor

Objects and Classes

Example:

This expression creates a
Scanner object in memory.

```
Scanner Input= new Scanner(System.in);
```

The object's memory address
is assigned to the Input
variable.



Uninitialized Local Reference Variables

- Reference variables can be declared without being initialized.

```
Rectangle box;
```

- This statement does not create a `Rectangle` object, so it is an uninitialized local reference variable.
- A local reference variable must reference an object before it can be used, otherwise a compiler error will occur.

```
box = new Rectangle();
```



More Examples

Car
– make – yearModel
+ setMake() + setYearModel() + getMake() + getYearModel()

CellPhone
– manufact : String – model : String – retailPrice : double
+ setManufact(man : String) : void + setModel(mod : String) : void + setRetailPrice(price : double) : void + getManufact() : String + getModel() : String + getRetailPrice() : double

Object Oriented Programming

Lecture 03

Constructors

Constructors

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.
- Constructors are used to perform operations at the time an object is created.
- Constructors typically initialize instance fields and perform other object initialization tasks.

Constructors

- Constructors have a few special properties that set them apart from normal methods.
 - Constructors have the same name as the class.
 - Constructors have no return type (not even `void`).
 - Constructors may not return any values.
 - Constructors are typically public.

Constructor for Rectangle Class

```
/**
 * Constructor
 * @param len The length of the rectangle.
 * @param w The width of the rectangle.
 */
public Rectangle(double len, double w)
{
    length = len;
    width = w;
}
```

Overloading Methods and Constructors

- Two or more methods in a class may have the same name as long as their parameter lists are different.
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Overloaded Method add

```
public int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

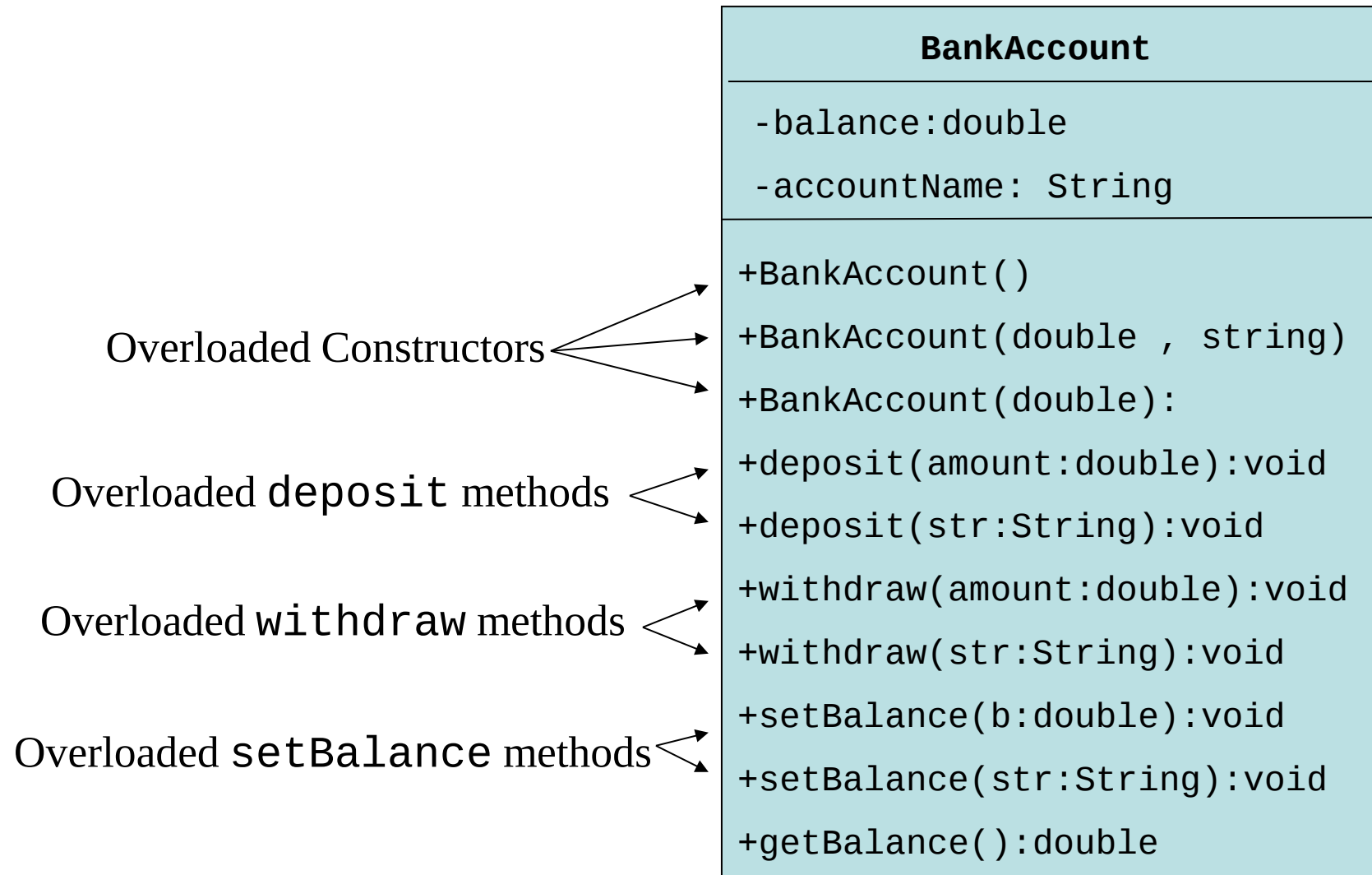
```
public String add (String str1, String str2)
{
    String combined = str1 + str2;
    return combined;
}
```

Rectangle Class Constructor Overload

```
Rectangle box1 = new Rectangle();
```

```
Rectangle box2 = new Rectangle(5.0, 10.0);
```

The BankAccount Example



The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, Java provides one when the class is compiled. The constructor that Java provides is known as the *default constructor*.
 - It sets all of the object's numeric fields to 0.
 - It sets all of the object's `boolean` fields to `false`.
 - It sets all of the object's reference variables to the special value *null*.

The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.

Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()  
{  
    length = 1.0;  
    width = 1.0;  
}
```

Object Oriented Programming

Lecture 04

Static Class members

Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

```
double val = Math.sqrt(25.0);
```



Class name

Static method

Static Fields

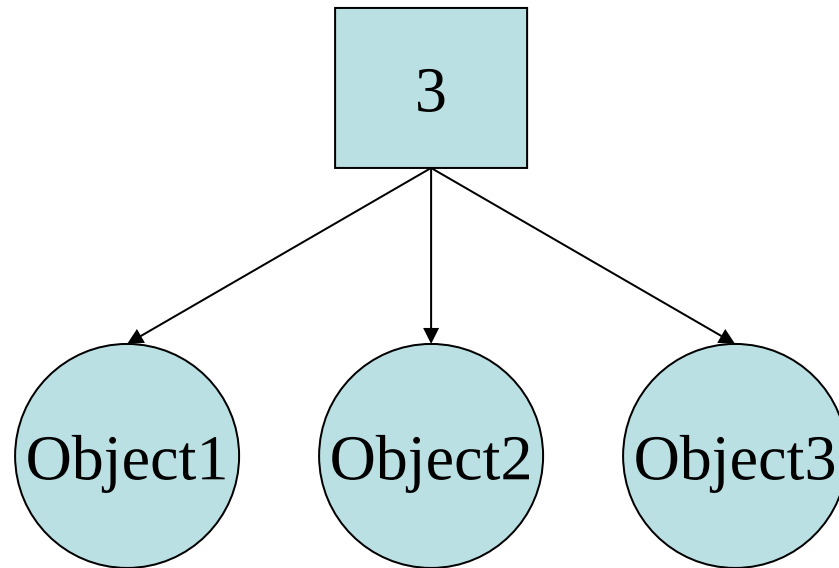
- Class fields are declared using the `static` keyword between the access specifier and the field type.

```
private static int instanceCount = 0;
```

- The field is initialized to 0 only once, regardless of the number of times the class is instantiated.
 - Primitive static fields are initialized to 0 if no initialization is performed.

Static Fields

instanceCount field
(static)



Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static double milesToKilometers(double miles)  
{...}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

```
double kilosPerMile = Metric.milesToKilometers(1.0);
```

Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

Object Oriented Programming

Lecture 05

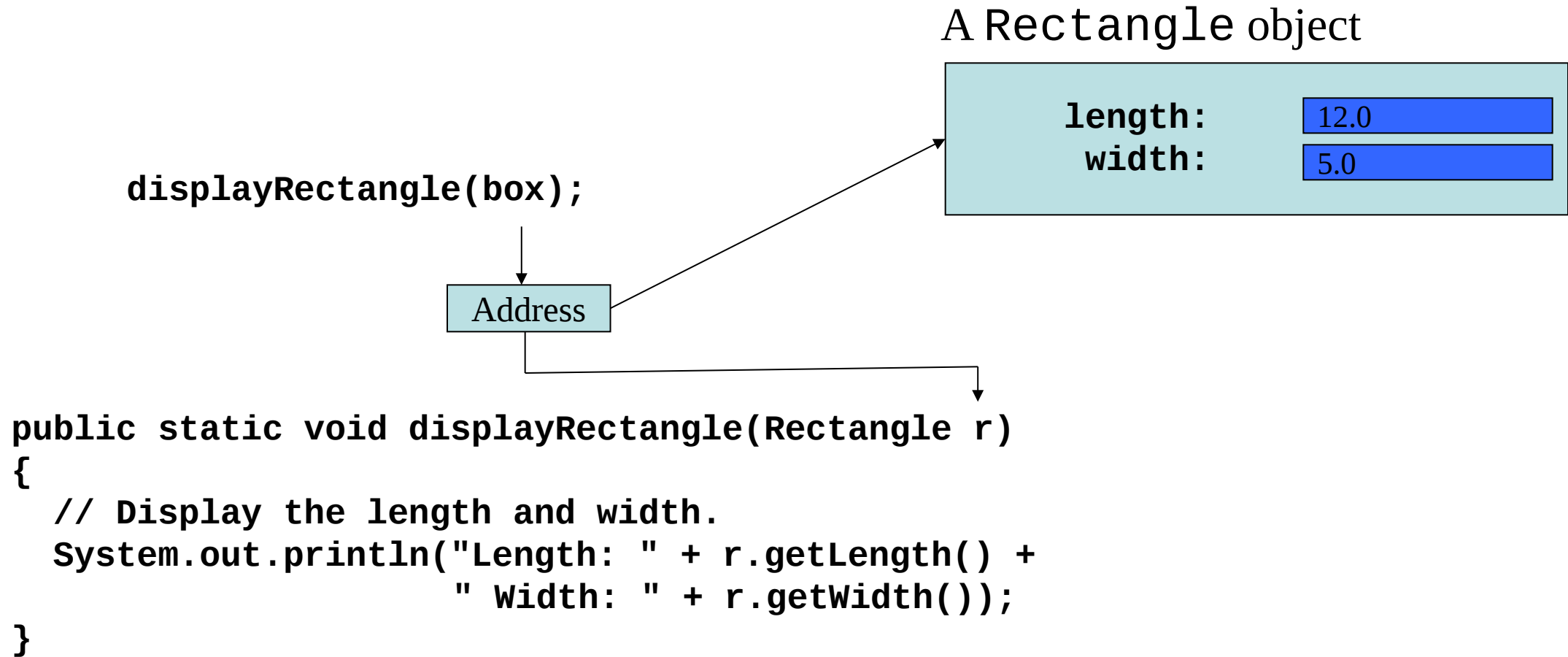
More about working With Objects

Passing , Returning, comparing and copying

Passing Objects as Arguments

- Objects can be passed to methods as arguments.
- Java passes all arguments *by value*.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A *copy* of the object is *not passed*, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.

Passing Objects as Arguments

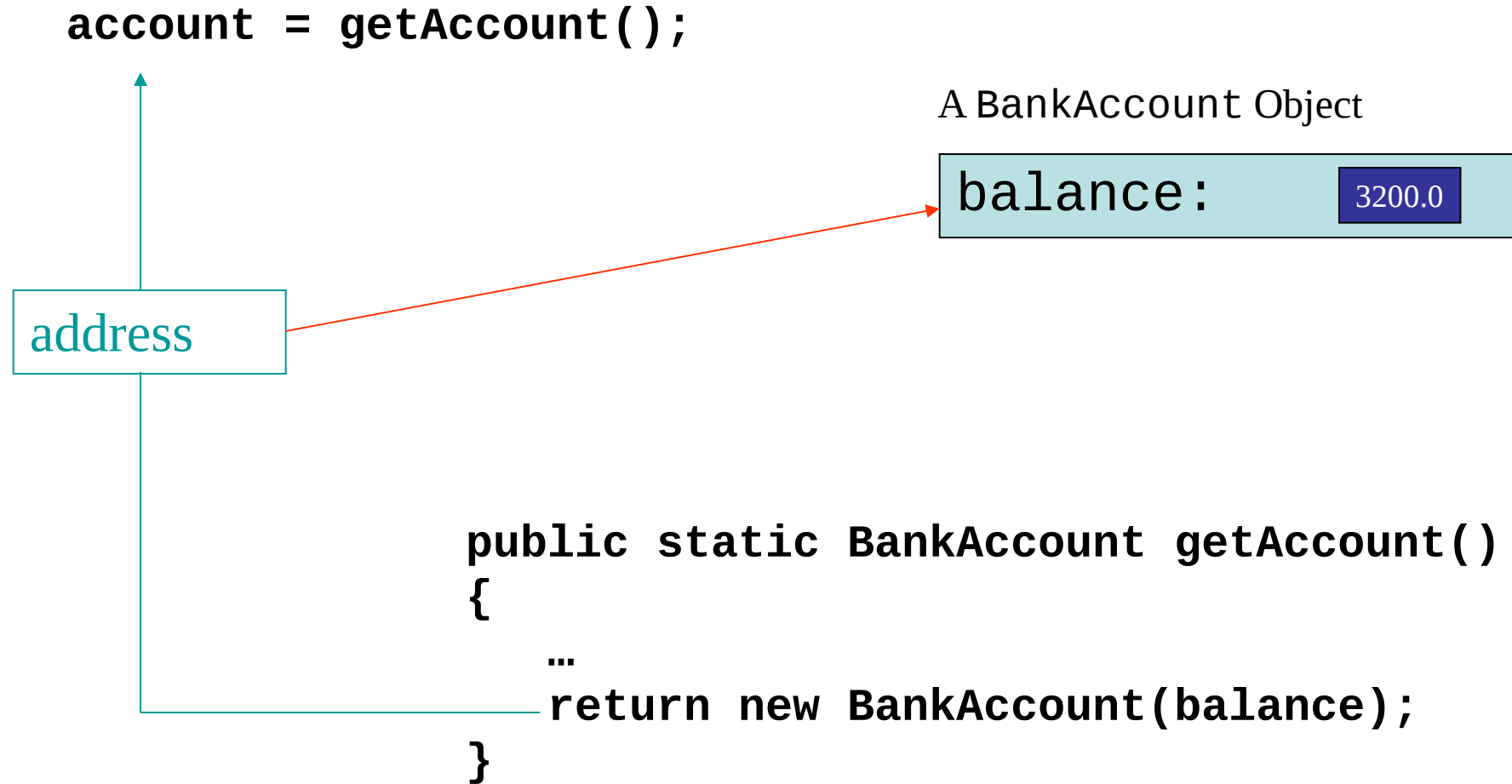


Returning Objects From Methods

- Methods are not limited to returning the primitive data types.
- Methods can return references to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, only its address.
- Method return type:

```
public static BankAccount getAccount()  
{  
    ...  
    return new BankAccount(balance);  
}
```

Returning Objects from Methods



Object Oriented Programming

Lecture 05 – Part 2

More about working With Objects

comparing and copying objects

Using The == operators with objects

- If we try the following:

```
Rectangle room1 = new Rectangle(10,50);
```

```
Rectangle room2 = new Rectangle(10,50);
```

```
if (room1 == room2) // This is a mistake.
```

```
    System.out.println("The objects are the same.");
```

```
else
```

```
    System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.

Methods That Copy Objects

- There are two ways to copy an object.
 - You cannot use the assignment operator to copy reference types
 - Reference only copy
 - This is simply copying the address of an object into another reference variable.
 - Deep copy (correct)
 - This involves creating a new instance of the class and copying the values from one object into the new object.

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object 2)
{
    symbol = object2.symbol;
    sharePrice = object2.sharePrice;
}
```

```
// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);
```

```
//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

Object Oriented Programming

Lecture 06 – Part 1

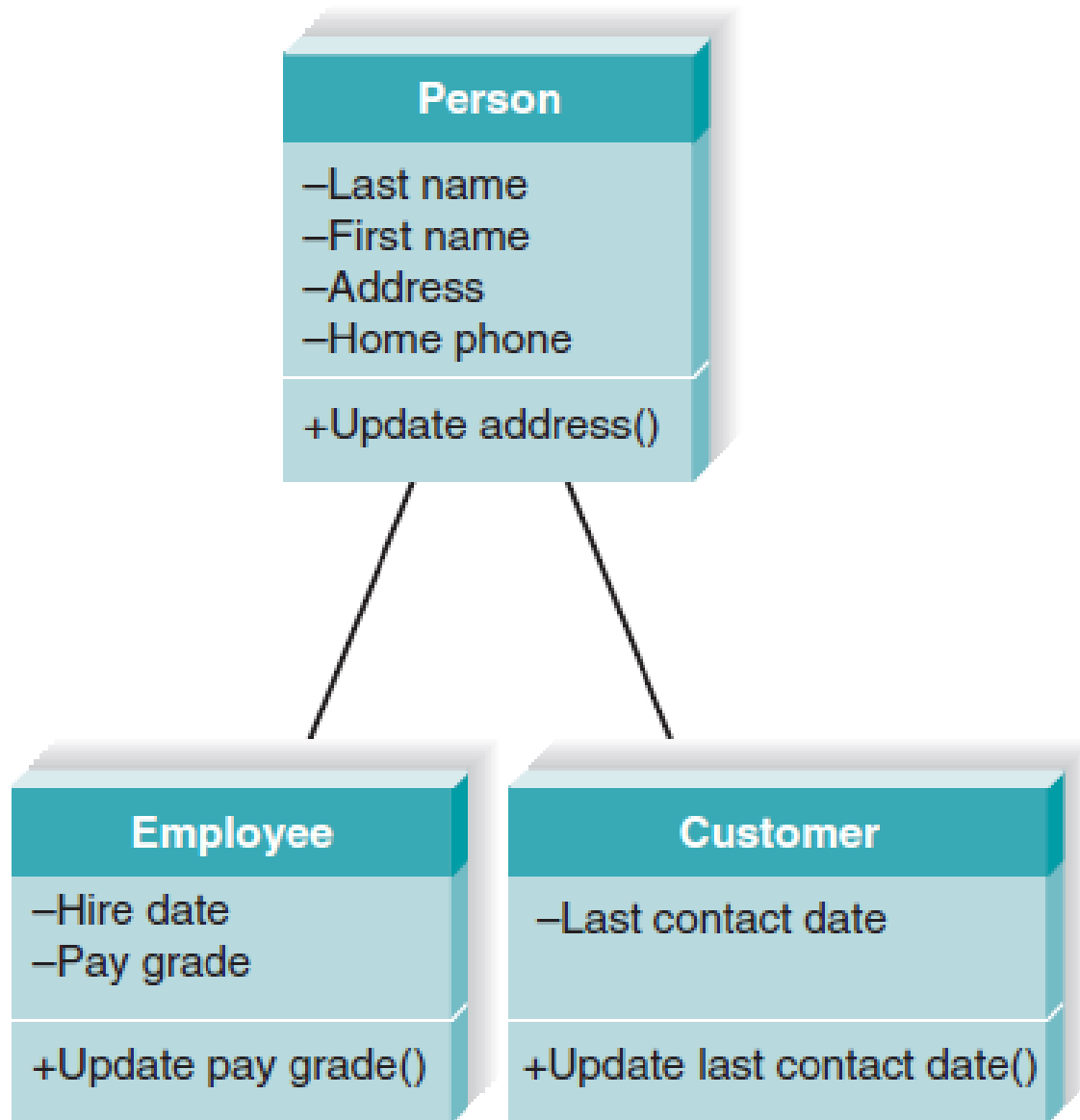
Inheritance and polymorphism

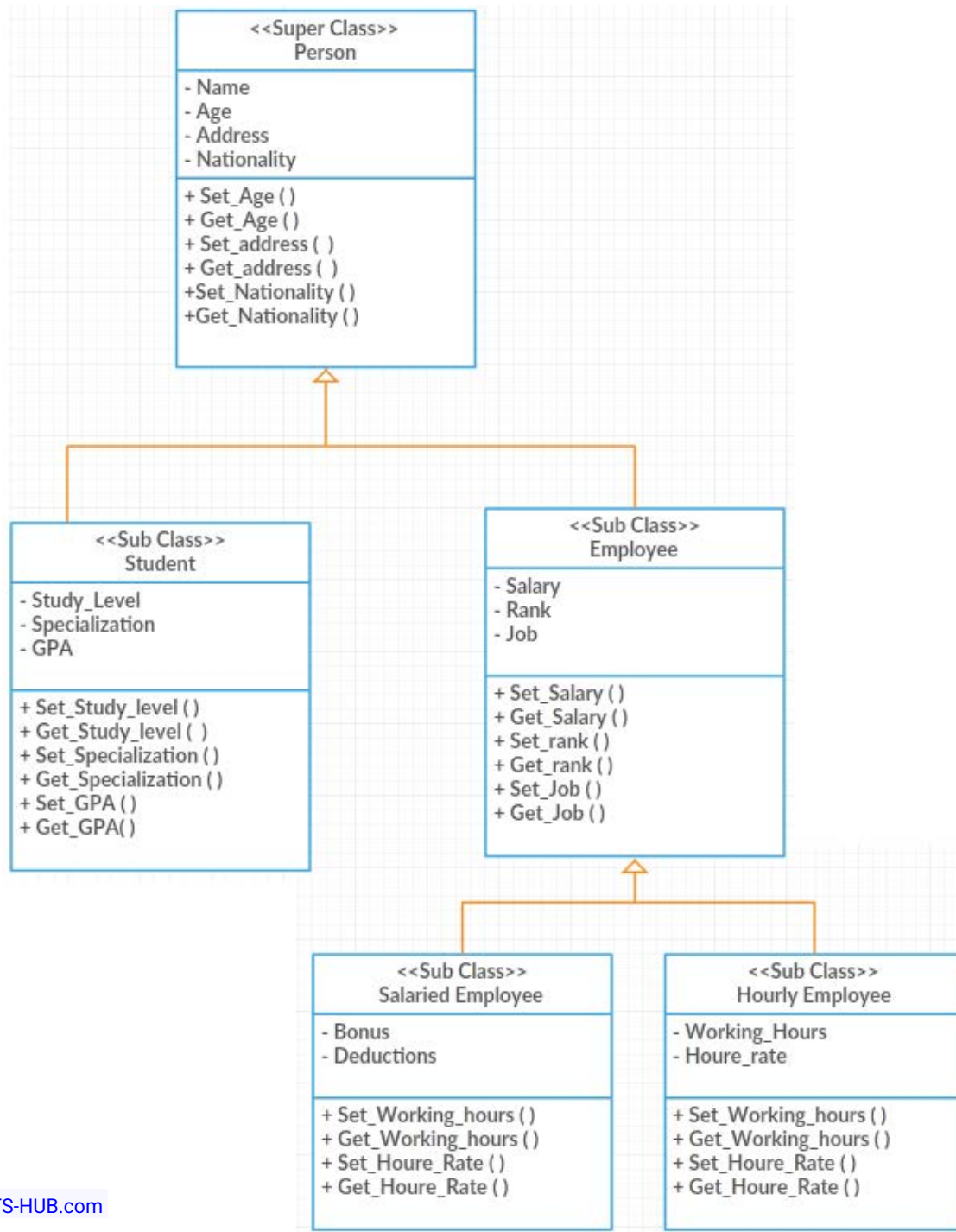
What is Inheritance?

Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term “Student” describes a very general type of Students with known characteristics.
- Post-Graduated Students and under-graduated students are Students
 - They share the general characteristics of an Student.
 - However, they have special characteristics of their own.
 - Post Graduated have an interesting research area.
 - Under Graduated have a Group no and class no .
- Post Graduated Students and under graduated students are specialized versions of a student.

With Inheritance





The “is a” Relationship

- The relationship between a superclass and an inherited class is called an “is a” relationship.
 - A post graduate student “is a” Student.
 - An Employee “is a” Person.
 - Salaried Employee “is a” Employee.
 - A car “is a” vehicle.
- A specialized object has:
 - all of the characteristics of the general object, plus
 - additional characteristics that make it special.
- In object-oriented programming, *inheritance* is used to create an “is a” relationship among classes.

The “is a” Relationship

- We can *extend* the capabilities of a class.
- Inheritance involves a superclass and a subclass.
 - The *superclass* is the general class and
 - the *subclass* is the specialized class.
- The subclass is based on, or extended from, the superclass.
 - Superclasses are also called *base classes*, and
 - subclasses are also called *derived classes*.
- The relationship of classes can be thought of as *parent classes* and *child classes*.

Object Oriented Programming

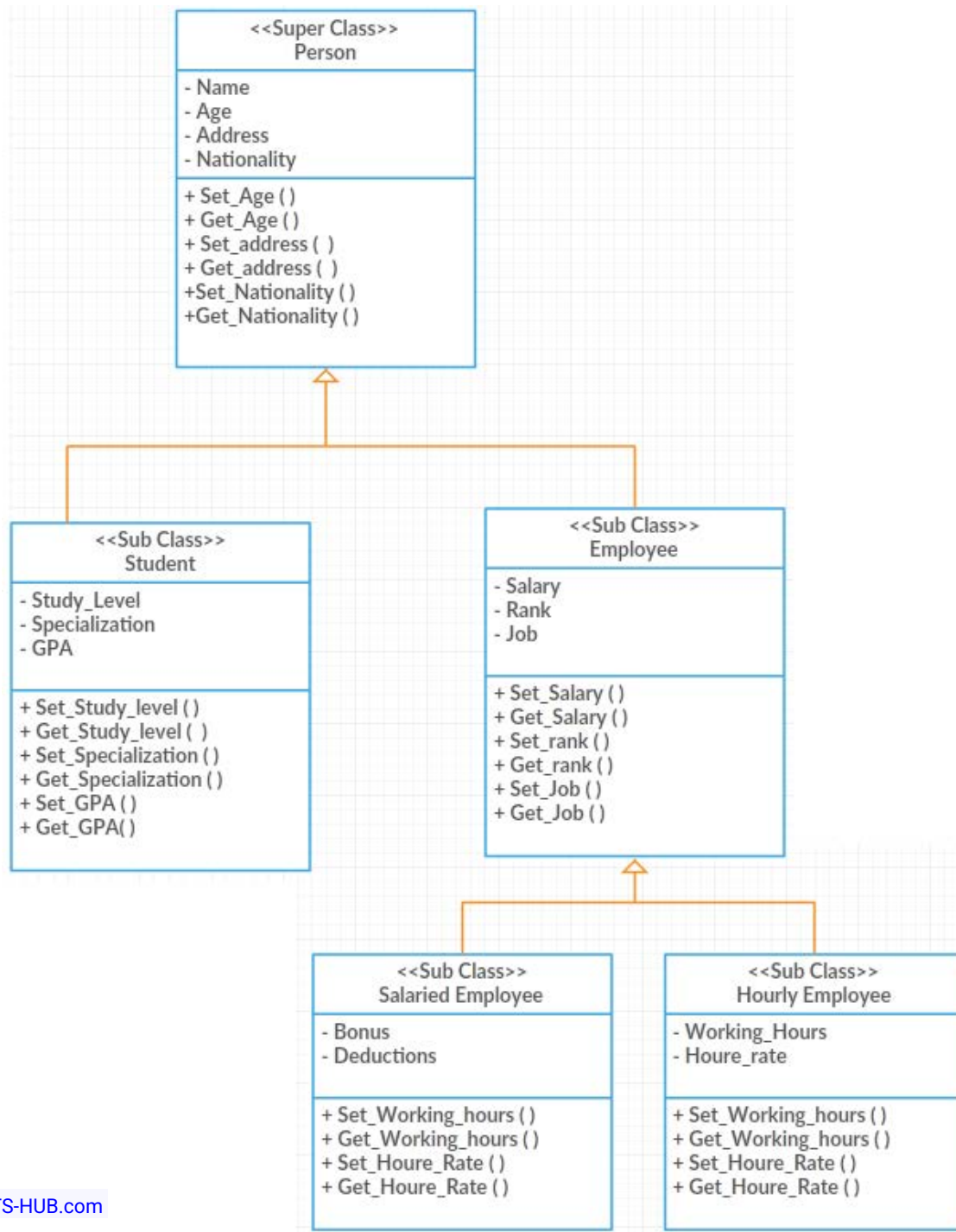
Lecture 06 – Part 2

Inheritance and polymorphism

Inheritance

- The subclass inherits fields and methods from the superclass without any of them being rewritten.
- New fields and methods may be added to the subclass.
- The Java keyword, *extends*, is used on the class header to define the subclass.

```
public class Employee extends Person
```



Inheritance, Fields and Methods

- Members of the superclass that are marked *private*:
 - are not inherited by the subclass,
 - exist in memory when the object of the subclass is created
 - may only be accessed from the subclass by public methods of the superclass.
- Members of the superclass that are marked *public*:
 - are inherited by the subclass, and
 - may be directly accessed from the subclass.

Inheritance, Fields and Methods

- When an instance of the subclass is created, the non-private methods of the superclass are available through the subclass object.

```
Employee emp1 = new Employee();  
Emp1.set_Age(30);  
System.out.println("Age = " + emp1.get_Age());
```

- Non-private methods and fields of the superclass are available in the subclass.

```
Set_Age(30);
```

Inheritance and Constructors

- Constructors are not inherited.
- When a subclass is instantiated, the superclass default constructor is executed first.
- The `super` keyword refers to an object's superclass.
- The superclass constructor can be explicitly called from the subclass by using the `super` keyword.

Calling The Superclass Constructor

- If a parameterized constructor is defined in the superclass,
 - the superclass must provide a no-arg constructor, or
 - subclasses must provide a constructor, and
 - subclasses must call a superclass constructor.
- Calls to a superclass constructor must be the first java statement in the subclass constructors.

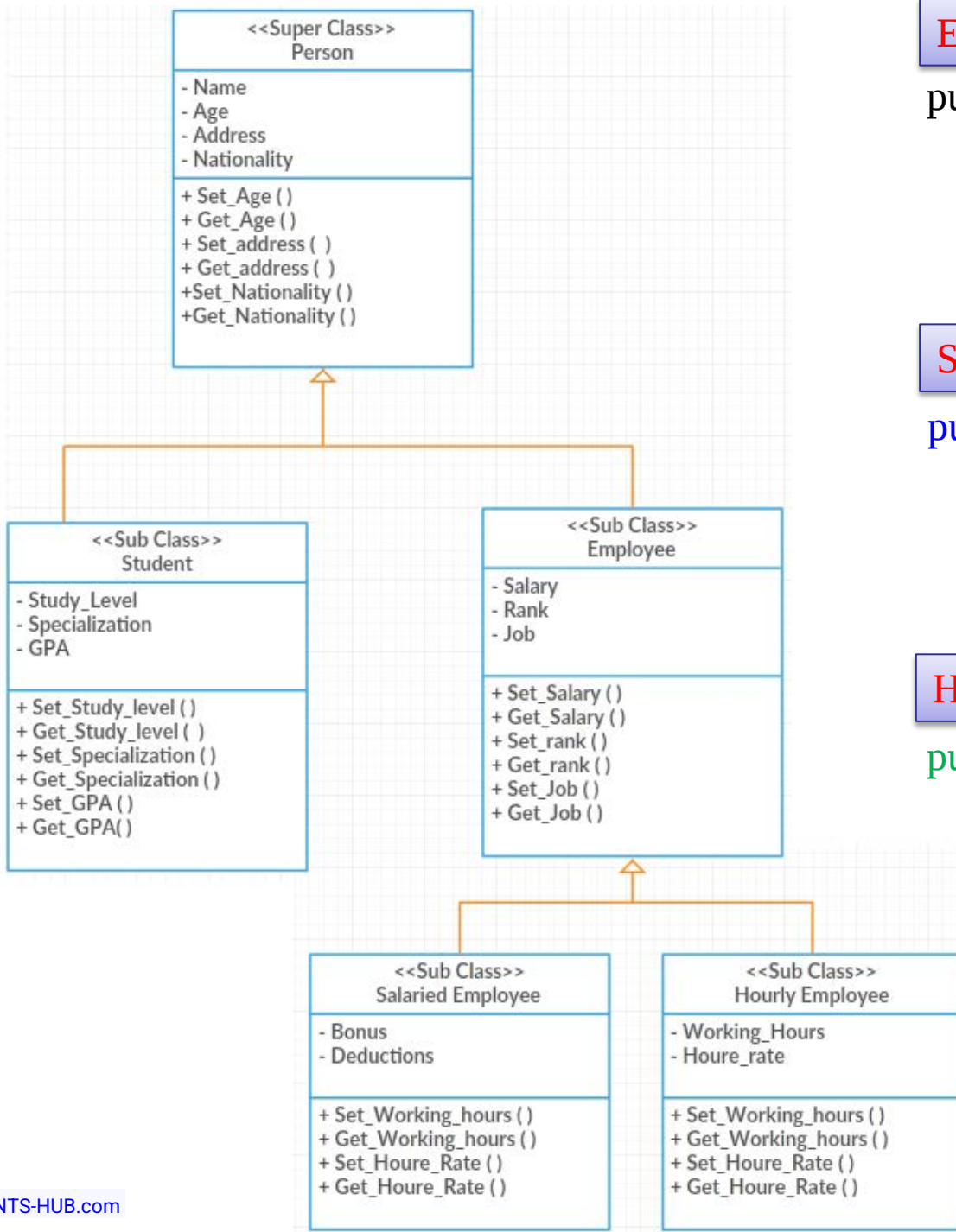
Object Oriented Programming

Lecture 06 – Part 3

Inheritance and polymorphism

Overriding Superclass Methods

- A subclass may have a method with the same signature as a superclass method.
- The subclass method overrides the superclass method.
- This is known as *method overriding*.
- A subclass method that overrides a superclass method must have the same signature as the superclass method.
- An object of the subclass invokes the subclass's version of the method, not the superclass's.
- The `@Override` annotation should be used just before the subclass method declaration.



Employee

```
public double get_salary( )
{
    return salary;
}
```

Salaried Employee

```
public double get_salary( )
{
    return salary + bonus - deductions ;
}
```

Hourly Employee

```
public double get_salary( )
{
    return working_hours * hours_rate ;
}
```

Overriding Superclass Methods

- An subclass method can call the overridden superclass method via the super keyword.

```
super.setScore(rawScore * percentage);
```

- There is a distinction between overloading a method and overriding a method.
- Overloading is when a method has the same name as one or more other methods, but with a different signature.
- Both overloading and overriding can take place in an inheritance relationship.
- Overriding can only take place in an inheritance relationship.

Object Oriented Programming

Lecture 06 – Part 4

Inheritance and Polymorphism

Final and Protected data members

Preventing a Method from Being Overridden

- The `final` modifier will prevent the overriding of a superclass method in a subclass.

```
public final void message()
```

- If a subclass attempts to override a final method, the compiler generates an error.
- This ensures that a particular superclass method is used by subclasses rather than a modified version of it.

Protected Members

- Using `protected` instead of `private` makes some tasks easier.
- Any class that is derived from the class, or is in the same package, has unrestricted access to the protected member.
- It is always better to make all fields `private` and then provide `public` methods for accessing those fields.
- If no access specifier for a class member is provided, the class member is given *package access* by default.
- Any method in the same package may access the member.

Protected Members

- Protected members of class:
 - may be accessed by methods in a subclass, and
 - by methods in the same package as the class.
- Java provides a third access specification, `protected`.
- A *protected* member's access is somewhere between *private* and *public*.

Package no1;

```
public class Shape
{
```

```
    Protected double height; // To hold height.
```

```
    Protected double width; //To hold width or base
```

```
    /**
```

```
     * The setValue method sets the data
     * in the height and width field.
```

```
    */
```

```
    public void setValues(double height, double width)
```

```
    {
```

```
        this.height = height;
```

```
        this.width = width;
```

```
    }
```

```
}
```

Package no2;

```
public class Rectangle extends Shape
{
```

```
    /**
```

```
     * The method returns the area
     * of rectangle.
```

```
    */
```

```
    public double getArea()
```

```
    {
```

```
        return height * width; //accessing protected members
```

```
    }
```

```
}
```

Object Oriented Programming

Lecture 07 – Part 1

Abstract Class and Interfaces

Abstract Classes

- An abstract class cannot be instantiated, but other classes are derived from it.
- An *Abstract class* serves as a superclass for other classes.
- The abstract class represents the generic or abstract form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

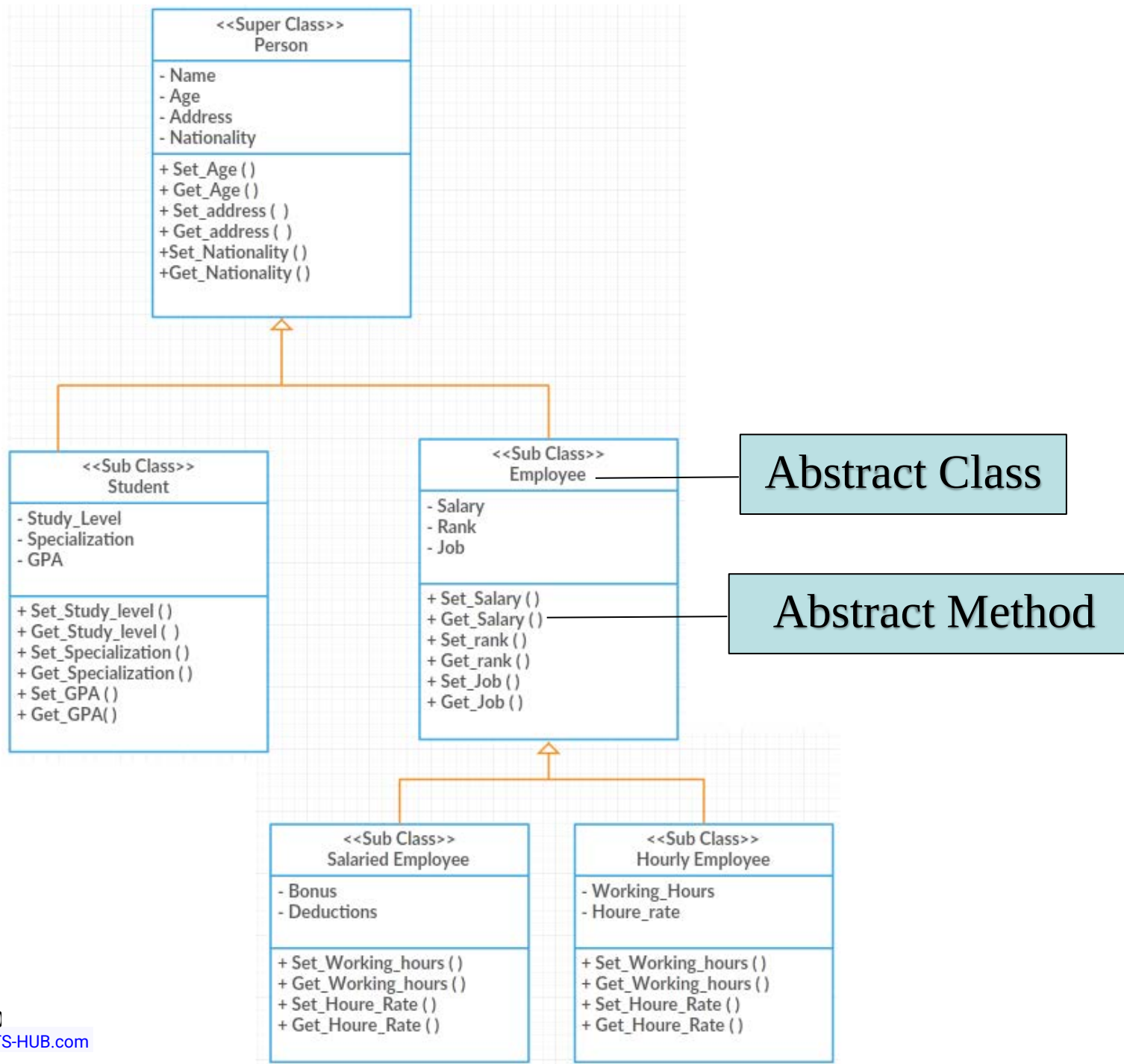
Abstract Methods

- An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass.
- An abstract method has no body and must be overridden in a subclass.

AccessSpecifier *abstract* ReturnType MethodName(ParameterList);

Ex: public abstract void GetSalary ();

- Any class that contains an abstract method is automatically abstract.
- Abstract methods are used to ensure that a subclass implements the method.
- If a subclass fails to override an abstract method, a compiler error will result.



Object Oriented Programming

Lecture 07 – Part 2

Abstract Class and Interfaces

Interfaces

- An *interface* is similar to an abstract class that has all abstract methods.
 - It cannot be instantiated, and
 - all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.
- The general format of an interface definition:

```
public interface InterfaceName  
{  
    (Method headers...)  
}
```

Class ABC

```
Public Void Method1 ( )  
{  
  
}
```

```
Public Void Method8 ( )  
{  
  
}
```

Implements

Contract

1- Method1()
2- Method2()

.

.

.

.

7- Method7()
8- Method8()

Interface

Interfaces

- A class can implement one or more interfaces
- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public interface RetailItem
{
    (Method headers...)
}
```

```
public class CD implements RetailItem
```

```
public class Book implements RetailItem
```

```

1  /**
2      RetailItem interface
3  */
4
5  public interface RetailItem
6  {
7      public double getRetailPrice();
8  }

1  /**
2      Compact Disc class
3  */
4
5  public class CompactDisc implements RetailItem
6  {
7      private String title;          // The CD's title
8      private String artist;         // The CD's artist
9      private double retailPrice;    // The CD's retail price
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52      public double getRetailPrice()
53      {
54          return retailPrice;
55      }

```

Fields in Interfaces

- An interface can contain field declarations:
 - all fields in an interface are treated as `final` and `static`.
- Because they automatically become `final`, you must provide an initialization value.

```
public interface Doable
{
    int FIELD1 = 1, FIELD2 = 2;
    (Method headers...)
}
```

- In this interface, `FIELD1` and `FIELD2` are `final static int` variables.
- Any class that implements this interface has access to these variables.

Implementing Multiple Interfaces

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
- When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the implements key word.

```
public class MyClass implements Interface1,  
                                Interface2,  
                                Interface3
```

Polymorphism with Interfaces

- Java allows you to create reference variables of an interface type.
- An interface reference variable can reference any object that implements that interface, regardless of its class type.

Polymorphism with Interfaces

- In the example code, two `RetailItem` reference variables, `item1` and `item2`, are declared.
- The `item1` variable references a `CompactDisc` object and the `item2` variable references a `DvdMovie` object.
- When a class implements an interface, an inheritance relationship known as *interface inheritance* is established.
 - a `CompactDisc` object *is a* `RetailItem`, and
 - a `DvdMovie` object *is a* `RetailItem`.

```
RetailItem item1 = new CompactDisc("Songs From the Heart","Billy Nelson",18.95);
```

```
RetailItem item2 = new DvdMovie("Planet X",102,22.95);
```

Polymorphism with Interfaces

- A reference to an interface can point to any class that implements that interface.
- You cannot create an instance of an interface.

```
RetailItem item = new RetailItem(); // ERROR!
```

- When an interface variable references an object:
 - only the methods declared in the interface are available,
 - explicit type casting is required to access the other methods of an object referenced by an interface reference.

Default Methods

- Beginning in Java 8, interfaces may have *default methods*.
- A default method is an interface method that has a body.

```
1 public interface Displayable
2 {
3     default void display()
4     {
5         System.out.println("This is the default display method.");
6     }
7 }
8
9 public class Person implements Displayable
10 {
11     private String name;
12
13     // Constructor
14     public Person(String n)
15     {
16         name = n;
17     }
18
19     public static void main(String[] args)
20     {
21         // Create an instance of the Person class.
22         Person p = new Person("Antonio");
23
24         // Call the object's display method.
25         p.display();
26     }
27 }
```

Object Oriented Programming

Lecture 08

Enumerated Types

Enumerated Types

- Known as an **enum**, requires declaration and definition like a class
- **Syntax:**

enum typeName { one or more enum constants }

- **Definition:**

enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

enum CarColor { RED, BLACK, BLUE, SILVER }

enum CarType { PORSCHE, FERRARI, JAGUAR }

- **Declaration:**

Day WorkDay; // creates a Day enum

- **Assignment:**

Day WorkDay = Day.WEDNESDAY;

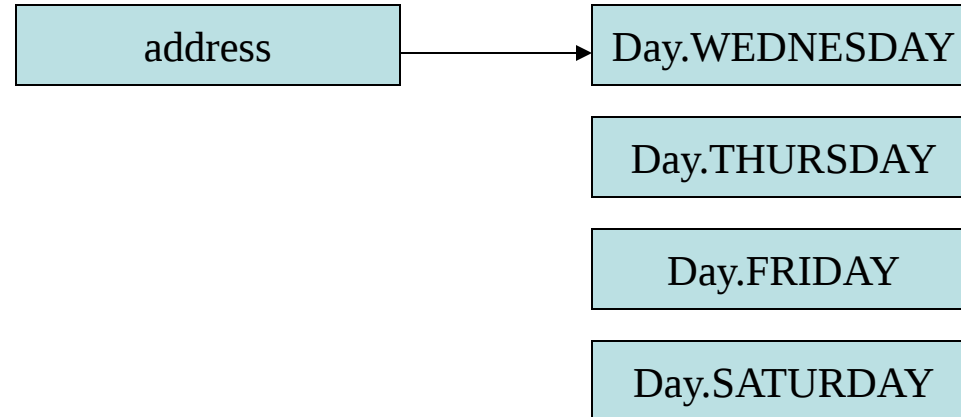
Enumerated Types

- An enum is a specialized class

Each are objects of type Day, a specialized class

```
Day workDay = Day.WEDNESDAY;
```

The workDay variable holds the address of the
Day.WEDNESDAY object



Enumerated Types - Methods

- `toString` – returns name of calling constant
- `ordinal` – returns the zero-based position of the constant in the enum. For example the ordinal for `Day.THURSDAY` is 4
- `equals` – accepts an object as an argument and returns true if the argument is equal to the calling enum constant
- `compareTo` - accepts an object as an argument and returns a negative integer if the calling constant's ordinal < than the argument's ordinal, a positive integer if the calling constant's ordinal > than the argument's ordinal and zero if the calling constant's ordinal == the argument's ordinal.

RegisterForm

- StdName: String
- StdGender : Gender
- CourseName: Course
- CrsSemester: Semester

Object Oriented Programming

Lecture 09

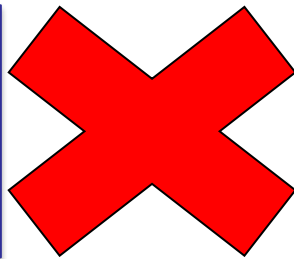
Exception Handling

Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Java allows you to create exception handlers.

```
int x = 10 , y = 0 ;
```

```
System.out.println (x/y);
```



Divide by Zero

Exception Classes

- An *exception handler* is a section of code that gracefully responds to exceptions.
- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

Handling Exceptions

- To handle an exception, you use a *try* statement.

```
try  
{  
    (try block statements...)  
}  
catch (ExceptionType ParameterName)  
{  
    (catch block statements...)  
}
```

- First the keyword `try` indicates a block of code will be attempted.

Handling Exceptions

- After the try block, a **catch** clause appears.
- A catch clause begins with the key word **catch**:
- **catch (*ExceptionType* *ParameterName*)**
 - *ExceptionType* is the name of an exception class and
 - *ParameterName* is a variable name which will reference the exception object if the code in the try block throws an exception.
- The code that immediately follows the catch clause is known as a *catch block*.
- The code in the catch block is executed if the try block throws an exception.

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
    File file = new File ("MyFile.txt");
    Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
    System.out.println("File not found.");
}
```

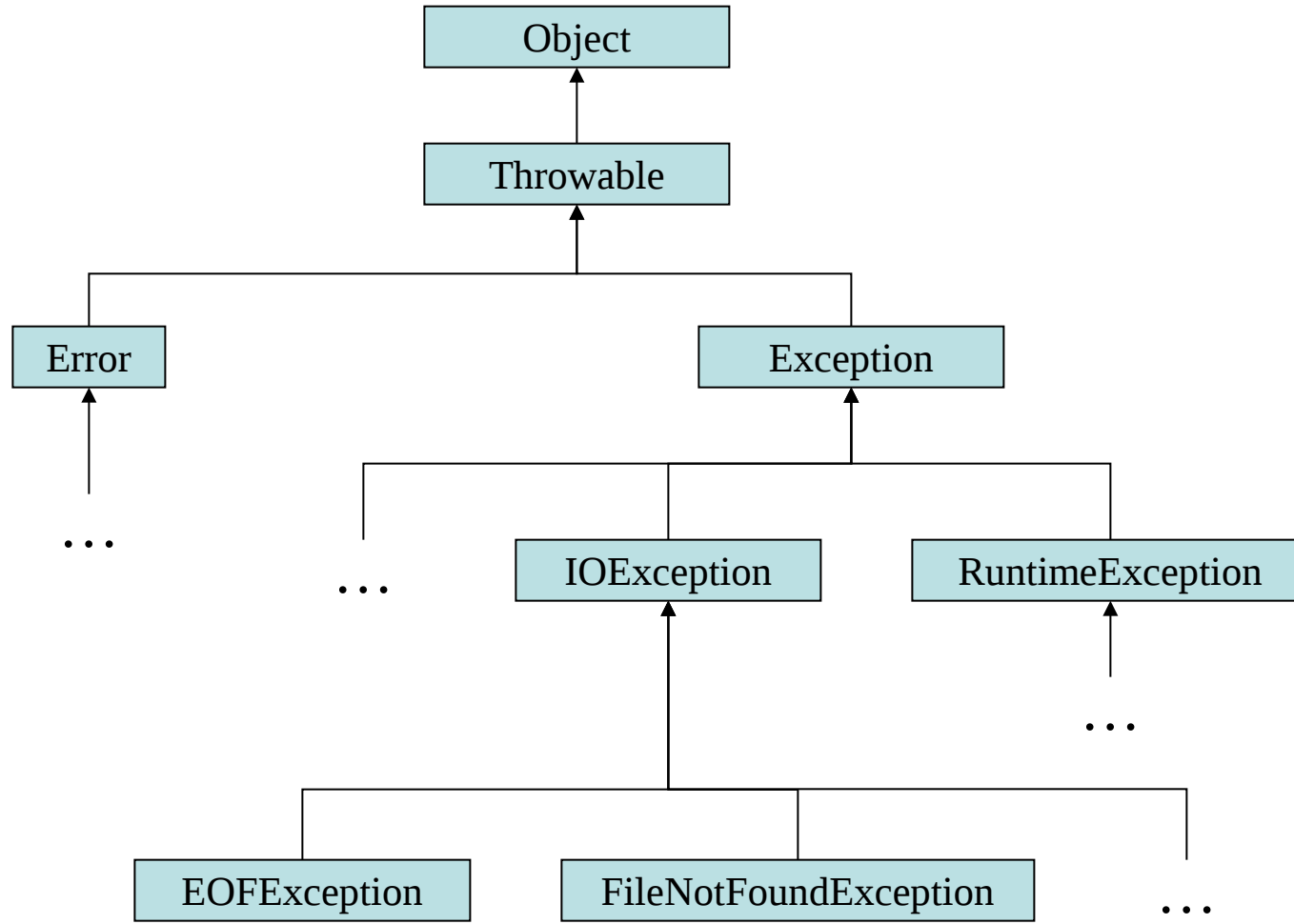
- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Polymorphic References To Exceptions

```
try
{
    number = Integer.parseInt(str);
}
catch (Exception e)
{
    System.out.println("The following error occurred: "
                       + e.getMessage());
}
```

- The Integer class's parseInt method throws a NumberFormatException object.
- The NumberFormatException class is derived from the Exception class.

Exception Classes



`FileNotFoundException` is **Exception**

`InputMismatchException` is **Exception**

`AnyException` is **Exception**

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found.
- The `catch` clauses must be listed from most specific to most general.

Exception Handlers

- There can be many polymorphic catch clauses.
- A try statement may have only one catch clause for each specific type of exception.

```
try
{
    number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
    System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

```
try
{
    number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
    System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The previous code could be rewritten to work, as follows, with no errors:

```
try
{
    number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
    System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e) //OK
{
    System.out.println("Bad number format.");
}
```

The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.

```
try
{
    (try block statements...)
}
catch (ExceptionType ParameterName)
{
    (catch block statements...)
}
finally
{
    (finally block statements...)
}
```

The *finally* Clause

- The *finally block* is one or more statements,
 - that are always executed after the try block has executed and
 - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

Throwing Exceptions

- You can write code that:
 - throws one of the standard Java exceptions, or
 - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

`throw new ExceptionType(MessageString);`

- The `throw` statement causes an exception object to be created and thrown.

Throwing Exceptions

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

Example:

```
if (Length == Width)
{
    throw new IllegalArgumentException( "In Rectangle ,The Length must be different from width.");
}
```

```
try
{
    int x , y=10;
    Scanner s= new Scanner(System.in);
    x= s.nextInt ( );
    if (x==0)
        throw new IllegalArgumentException("Must be more than 0");
    System.out.println(y/x);
}

catch(ArithmeticException e)
{
    System.out.println("Error");
}
catch(IllegalArgumentException e2){
    System.out.println("wrong value");
}
catch (InputMismatchException e3)
{
    System.out.println("Enter only numeric value");
}
    System.out.println("Final");

}
```

Object Oriented Programming

Lecture 10

The ArrayList Class

The ArrayList Class

- Similar to an array, an `ArrayList` allows object storage
- Unlike an array, an `ArrayList` object:
 - Automatically expands when a new item is added
 - Automatically shrinks when items are removed
- Requires:

```
import java.util.ArrayList;
```

Creating an ArrayList

```
ArrayList<String> nameList = new ArrayList<String>();
```

Two red arrows are present. One arrow points from below the first 'String' (in the generic type parameter) up to it. The other arrow points from below the second 'String' (in the constructor argument) up to it.

Notice the word `String` written inside angled brackets `<>`

This specifies that the `ArrayList` can hold `String` objects.

If we try to store any other type of object in this `ArrayList`, an error will occur.

Using an ArrayList

- To populate the `ArrayList`, use the `add` method:
 - `nameList.add("James");`
 - `nameList.add("Catherine");`
- To get the current size, call the `size` method
 - `nameList.size();` *// returns 2*
- To access items in an `ArrayList`, use the `get` method
 - `nameList.get(1);`

Using an ArrayList

- The `ArrayList` class's `toString` method returns a string representing all items in the `ArrayList`

```
System.out.println(nameList);
```

This statement yields :

```
[ James, Catherine ]
```

- The `ArrayList` class's `remove` method removes designated item from the `ArrayList`

```
nameList.remove(1);
```

This statement removes the second item.

Using an ArrayList

- The `ArrayList` class's `add` method with one argument adds new items to the end of the `ArrayList`
- To insert items at a location of choice, use the `add` method with two arguments:

```
nameList.add(1, "Mary");
```

This statement inserts the `String` "Mary" at index 1

- To replace an existing item, use the `set` method:

```
nameList.set(1, "Becky");
```

This statement replaces "Mary" with "Becky"

Using an ArrayList

- An `ArrayList` has a capacity, which is the number of items it can hold without increasing its size.
- The default capacity of an `ArrayList` is 10 items.
- To designate a different capacity, use a parameterized constructor:

```
ArrayList<String> list = new ArrayList<String>(100);
```

Using an ArrayList

- You can store any type of *object* in an ArrayList

```
ArrayList<BankAccount> accountList =  
    new ArrayList<BankAccount>();
```



This creates an ArrayList that can hold
BankAccount objects.

Using an ArrayList

```
// Create an ArrayList to hold BankAccount objects.
ArrayList<BankAccount> list = new ArrayList<BankAccount>();

// Add three BankAccount objects to the ArrayList.
list.add(new BankAccount(100.0));
list.add(new BankAccount(500.0));
list.add(new BankAccount(1500.0));

// Display each item.
for (int index = 0; index < list.size(); index++)
{
    BankAccount account = list.get(index);
    System.out.println("Account at index " + index +
        "\nBalance: " + account.getBalance());
}
```

Object Oriented Programming with Java

Revision On All OOP Concepts

